Csapdák és buktatók a modern C++-ban

Porkoláb Zoltán
Ericsson Magyarország Kft.
http://gsd.web.elte.hu/
zoltan.porkolab@ericsson.com

Az előadó

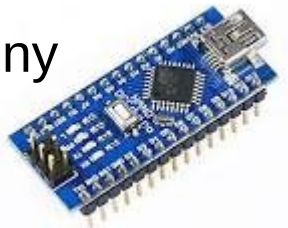
- 1984 óta programoz C-ben, 1988 óta C++-ban
- 1990 óta tanít C ill. C++ nyelvet
- Stroustrup: The C++ Programming Language 3rd spec ed. 2001 fordítása
- Az Ericsson Magyarország statikus analízis csapatának vezetője
- Az ELTE Informatikai karának oktatója
- CodeChecker, CodeCompass, Templight eszközök

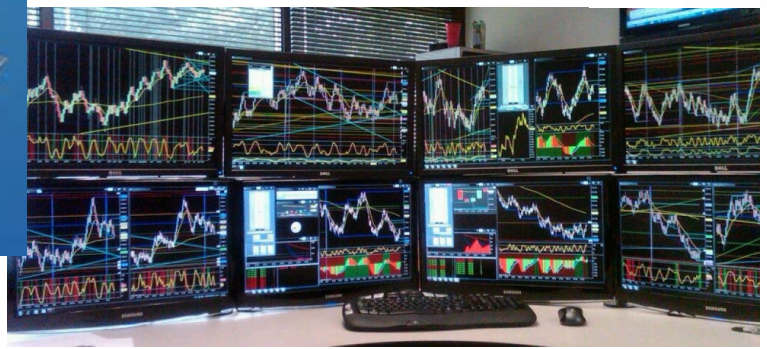
Tartalom

- A C++ nyelv fejlődése
- Move-szemantika
- Smart pointers
- Párhuzamos STL
- Összefoglalás

A C++ nyelv megkerülhetetlen

- 2022-ben a 3-4 a Tiobe indexen
- Energia-felhasználása kb. Java 1/3-a, Python 1/20-a
- Memória-felhasználása hasonlóan alacsony

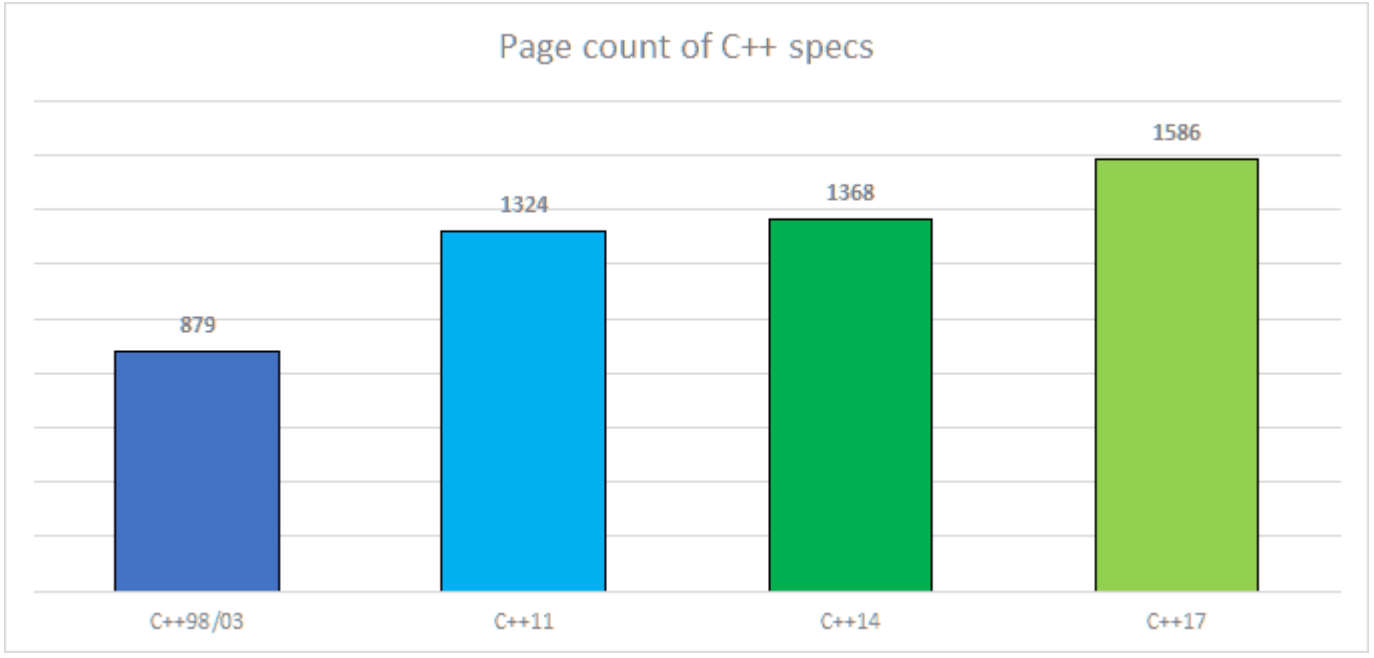




A C++ nyelv fejlődése

- 1980 C with Classes
- 1998 C++ standard ISO/IEC 14882:1998
- 2003 Kisebb javítások a szabványban
- 2011 Teljes nyelvmegújítás: C++11, Memória model, variadic templates, ...
- 2014 Kisebb módosítások: generikus lambda/capture, constexpr bővítés, ...
- 2017 Nagyobb bővítés: CTAD, ParSTL, Structured binding, új könyvtárak, ...
- 2020 Ismét nagyobb bővítés: Modulok, Concept-ek, …
- 2023 Még több constexpr, ranges, stacktrace, t[i,j,k], ...

A C++ nyelv fejlődése



(by Bartłomiej Filipek: How to Stay Sane With Modern C++)
https://dzone.com/articles/how-to-stay-sane-with-modern-c

A C++ nyelv fejlődése

- A szabványbizottság a legjobb szakértőkből áll
- A szabványbizottságban ott vannak a fordító- és könyvtár-írók
- A szabványbizottság csak kipróbált, implementált elemeket fogad be
- A szabványbizottság vitára bocsájtja a draft-okat
- A szabványbizottság biztosítja a visszafelé-kompatibilitást

A C++ nyelv fejlődése

- A szabványbizottság a legjobb szakértőkből áll
- A szabványbizottságban ott vannak a fordító- és könyvtár-írók
- A szabványbizottság csak kipróbált, implementált elemeket fogad be
- A szabványbizottság vitára bocsájtja a draft-okat
- A szabványbizottság biztosítja a visszafelé-kompatibilitást
- A szabványbizottság nem hozhat olyan szabályt, amely megakadályozná a programozót, hogy lábon lője magát!*

* http://thbecker.net/articles/rvalue_references/section_04.html

Visszafelé kompatibilitás

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

Visszafelé kompatibilitás

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++98  vec.cpp &&  ./a.out
S() copyCtr copyCtr copyCtr copyCtr copyCtr
S() copyCtr copyCtr copyCtr copyCtr copyCtr copyCtr
1 1 1 1 1 2
```

Visszafelé kompatibilitás

```cpp
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
vector (size_type n,
        const value_type& val = value_type(),
        const allocator_type& alloc = allocator_type());
```

Visszafelé kompatibilitás

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

Visszafelé kompatibilitás

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11  vec.cpp &&  ./a.out
S() S() S() S() S() S()
copyCtr copyCtr copyCtr copyCtr copyCtr copyCtr
1 2 3 4 5 6
```

Visszafelé kompatibilitás

```cpp
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
vector (size_type n,
        const value_type& val = value_type(),
        const allocator_type& alloc = allocator_type());
```

Visszafelé kompatibilitás

```cpp
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
vector (size_type n,
        const value_type& val,
        const allocator_type& alloc = allocator_type());
explicit vector (size_type n);   // since C++11 until C++14
```

Move szemantika

- Másolás helyett használjuk fel/fosszuk ki az eltűnő temporálisokat
- Hagyjuk a kifosztott objektumot, helyes, nem-specifikált, destruálható állapotban
- Az összes szabványos könyvtárt továbbfejlesztették a move-szemantikához
- Komoly sebességelőnyt remélünk (kivéve RVO esetén)

Move szemantika

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs)         { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs)      { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

Move szemantika

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs)         { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs)      { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11  vec.cpp &&  ./a.out
S() S() S() S() S() S()
moveCtr copyCtr copyCtr copyCtr copyCtr copyCtr
1 2 3 4 5 6
```

Move szemantika

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs)         { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs)      { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S> sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11  vec.cpp &&  ./a.out
S() S() S() S() S() S()
moveCtr copyCtr copyCtr copyCtr copyCtr copyCtr
1 2 3 4 5 6
```

Kivétel-garanciák

A szabványos könyvtár az alábbi garanciákat adja:

- Alap (basic) garancia: nem következik be erőforrás/memória elszivárgás
- Erős (strong) garancia: a művelet atomikus
  - Vector push_back(), Assoc. konténerek insert()-je, stb...
- Nincsen kivétel (nothrow): a művelet nem dobhat kivételt
  - Vector pop_back(), Assoc. Konténerek erase()-e, swap()

Move szemantika

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs) noexcept        { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) noexcept { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

Move szemantika

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs) noexcept        { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) noexcept { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

```
$ g++ -std=c++11  vec.cpp &&  ./a.out
S() S() S() S() S() S()
moveCtr moveCtr moveCtr moveCtr moveCtr moveCtr
1 2 3 4 5 6
```

Figyelmeztetés

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs) noexcept        { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) noexcept { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;

int main()
{
    std::vector<S>  sv(5);
    sv.push_back(S());

    for (std::size_t i = 0; i < sv.size(); ++i)
        std::cout << sv[i].a << " ";
    std::cout << std::endl;
}
```

EZT A KONSTRUKCIÓT
NE HASZNÁLD KONKURENS KÖRNYEZETBEN!

```
$ g++ -std=c++11 vec.cpp && ./a.out
S() S() S() S() S() S()
moveCtr moveCtr moveCtr moveCtr moveCtr moveCtr
1 2 3 4 5 6
```

Move algoritmus

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs) noexcept        { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) noexcept { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;


int f(std::list<S>& from, std::vector<S>& to)
{
    std::move( from.begin(), from.end(), to.begin());
}
```

Move algoritmus

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs) noexcept        { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) noexcept { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;


int f(std::list<S>& from, std::vector<S>& to)
{
    std::move( from.begin(), from.end(), to.begin());
}
```

```
$ g++ -std=c++11  vec.cpp &&  ./a.out
move= move= move= move= move=
```

Move algoritmus

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs) noexcept        { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) noexcept { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;
bool operator<(const S& x, const S& y) { return x.a < y.a; }

int f(std::set<S>& from, std::vector<S>& to)
{
    std::move( from.begin(), from.end(), to.begin());
}
```

Move algoritmus

```
#include <iostream>
#include <vector>

struct S
{
    S() { a = ++cnt; std::cout << "S() ";  }
    S(const S& rhs)    { a = rhs.a; std::cout << "copyCtr ";  }
    S(S&& rhs) noexcept        { a = rhs.a; std::cout << "moveCtr "; }
    S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy= "; return *this; }
    S& operator=(S&& rhs) noexcept { a = rhs.a; std::cout << "move= "; return *this; }
    int a;
    static int cnt;
};
int S::cnt = 0;
bool operator<(const S& x, const S& y) { return x.a < y.a; }

int f(std::set<S>& from, std::vector<S>& to)
{
    std::move( from.begin(), from.end(), to.begin());
}
```

```
$ g++ -std=c++11  vec.cpp &&  ./a.out
copy= copy= copy= copy= copy=
```

Az std::set konténer

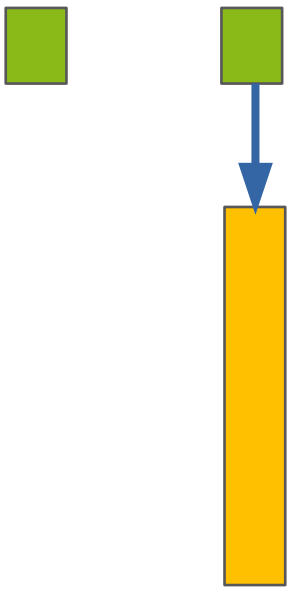
- C++11 előtt
  - Set::iterator kétirányú (bidirectional) iterátor
  - Set::const_iterator konstans kétirányú (bidirectional) iterátor
- C++11 után
  - Set::iterator konstans kétirányú (bidirectional) iterátor
  - Set::const_iterator konstans kétirányú (bidirectional) iterátor
- A konstans iterátorral meglátogatott elemek nem move-olhatóak
- A move csendben másolássá minősül vissza

Move szemantika

- Másolás helyett használjuk fel/fosszuk ki az eltűnő temporálisokat
- Hagyjuk a kifosztott objektumot, helyes, nem-specifikált, destruálható állapotban
- Az összes szabványos könyvtárt továbbfejlesztették a move-szemantikához
- Komoly sebességelőnyt remélünk (kivéve RVO esetén)

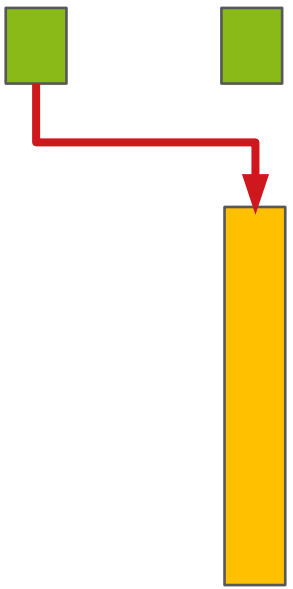
PIMPL unique_ptr

```
class S
{
public:
    S();
    ~S();
    S(const S& rhs);
    S(S&& rhs) noexcept;
    S& operator=(const S& rhs);
    S& operator=(S&& rhs) noexcept;
private:
    struct Ximpl;
    std::unique_ptr<XImpl> pImpl;
};
```



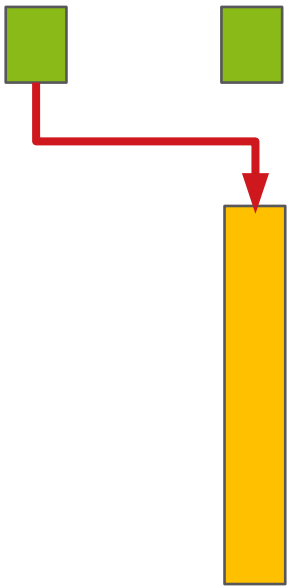
PIMPL unique_ptr

```
class S
{
public:
    S();
    ~S();
    S(const S& rhs);
    S(S&& rhs) noexcept;
    S& operator=(const S& rhs);
    S& operator=(S&& rhs) noexcept;
private:
    struct Ximpl;
    std::unique_ptr<XImpl> pImpl;
};
```



PIMPL unique_ptr

```
class S
{
public:
    S();
    ~S();
    S(const S& rhs);
    S(S&& rhs) noexcept;
    S& operator=(const S& rhs);
    S& operator=(S&& rhs) noexcept;
private:
    struct Ximpl;
    std::unique_ptr<XImpl> pImpl;
};
```



VALID?

Okos mutatók

- A RAII (Resource Acquisition Is Initialization) implementálása memória esetére
- Az objektum élettartama definiálja az erőforrás lefoglalását
- Szabványos okos mutatók: unique_ptr és a shared_ptr/weak_ptr
- A különbség a tulajdonosság implementálása
- Egyedi objektumra vagy tömbre mutat (és ezek nem keverednek)
- Factory függvények (make_unique() és make_shared())
- Deleter paraméter

Polimorfizmus

```
struct Base {
  virtual void f() { std::cout << "Base::f\n"; }
  virtual ~Base()  { std::cout << "Base::~Base()\n"; }
};
struct Derived : Base {
  virtual void f() override   { std::cout << "Derived::f\n"; }
  virtual ~Derived() override { std::cout << "Derived::~Derived()\n"; }
};

void g() {
  Derived *dp = new Derived{};
  Base    *bp = dp;

  delete  bp;   // Derived::~Derived()
};
```

Polimorfizmus

```
struct Base {
  virtual void f() { std::cout << "Base::f\n"; }
  virtual ~Base()  { std::cout << "Base::~Base()\n"; }
};
struct Derived : Base {
  virtual void f() override   { std::cout << "Derived::f\n"; }
  virtual ~Derived() override { std::cout << "Derived::~Derived()\n"; }
};

void g() {
  auto dp = std::make_shared<Derived>();
  std::shared_ptr<Base> bp = dp;


};   // Derived::~Derived()
```

Polimorfizmus

```
struct Base {
  virtual void f() { std::cout << "Base::f\n"; }
  virtual ~Base()  { std::cout << "Base::~Base()\n"; }
};
struct Derived : Base {
  virtual void f() override   { std::cout << "Derived::f\n"; }
  virtual ~Derived() override { std::cout << "Derived::~Derived()\n"; }
};

void g() {
  auto dp = std::make_shared<Derived>();
  std::shared_ptr<Base> bp = dp;  // deleter is copied


};   // Derived::~Derived()
```

Polimorfizmus

```
struct Base {
  virtual void f() { std::cout << "Base::f\n"; }
  virtual ~Base()  { std::cout << "Base::~Base()\n"; }
};
struct Derived : Base {
  virtual void f() override   { std::cout << "Derived::f\n"; }
  virtual ~Derived() override { std::cout << "Derived::~Derived()\n"; }
};

void g() {
  auto dp = std::make_unique<Derived>();
  std::unque_ptr<Base> bp = std::move(dp);  // Pointer is moved


};   // Derived::~Derived()
```

Polimorfizmus

```
struct Base {
  virtual void f() { std::cout << "Base::f\n"; }
  virtual ~Base()  { std::cout << "Base::~Base()\n"; }
};
struct Derived : Base {
  virtual void f() override   { std::cout << "Derived::f\n"; }
  virtual ~Derived() override { std::cout << "Derived::~Derived()\n"; }
};

void g() {
  auto dp = std::make_unique<Derived>();
  std::unque_ptr<Base> bp = std::move(dp);  // Pointer is moved


};   // Base::~Base()
```

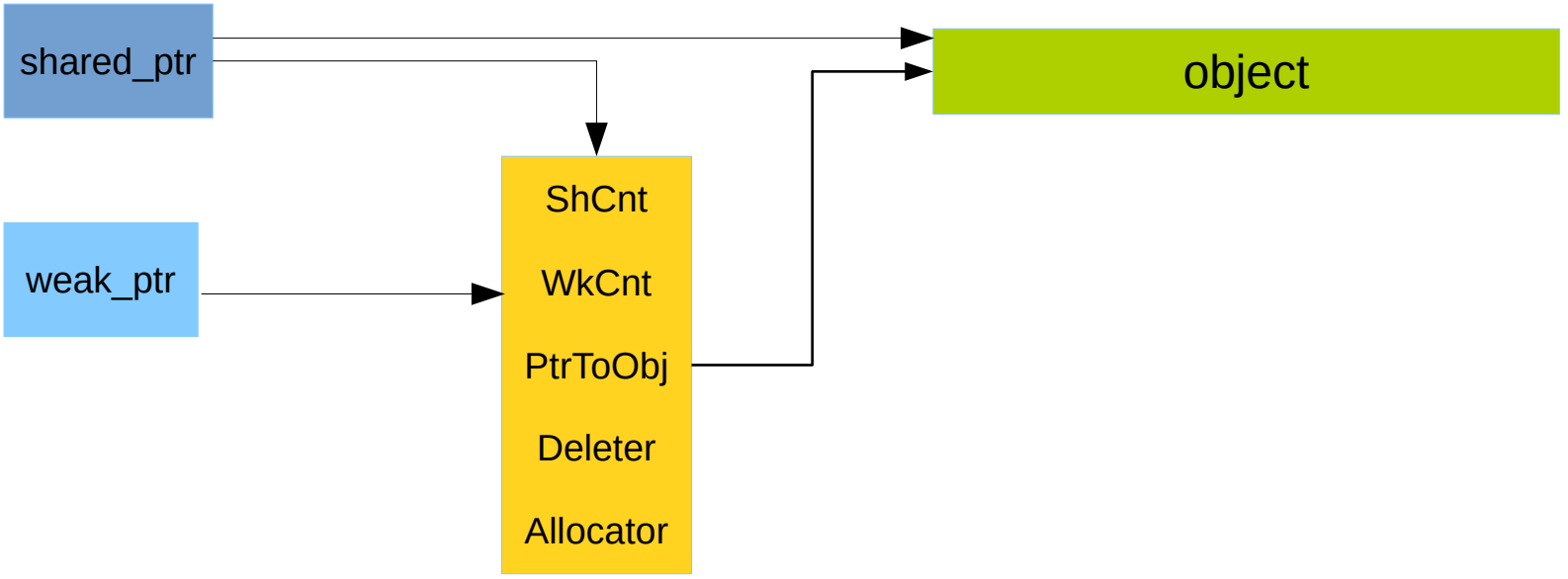
Polimorfizmus

```
struct Base {
  virtual void f() { std::cout << "Base::f\n"; }
  virtual ~Base()  { std::cout << "Base::~Base()\n"; }
};
struct Derived : Base {
  virtual void f() override   { std::cout << "Derived::f\n"; }
  virtual ~Derived() override { std::cout << "Derived::~Derived()\n"; }
};

void g() {
  auto dp = std::make_unique<Derived>();
  std::unque_ptr<Base> bp = std::move(dp);  // Pointer is moved, deleter is not copied


};   // Base::~Base()
```

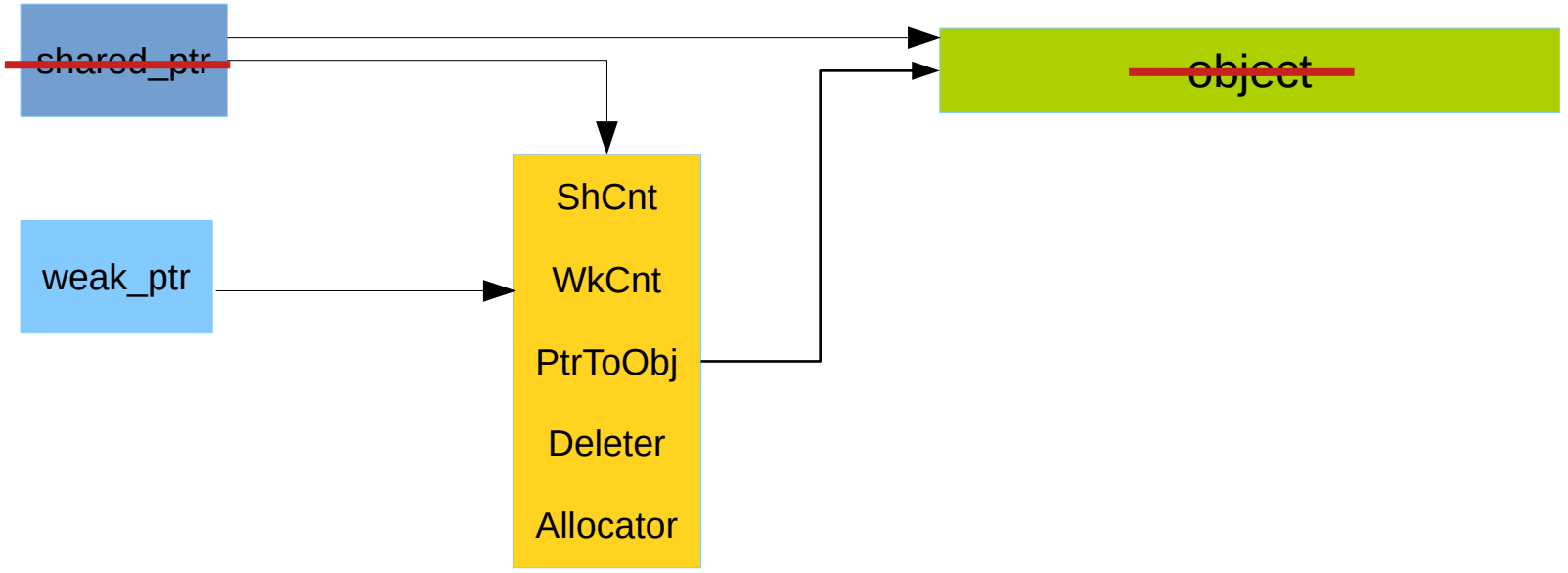
Üres bázis optimalizálás

```
template<class _T, class _DeleterT = std::default_delete<_T>>
class unique_ptr
{
public:
  // public interface...
private:
  // using empty base class optimization to save space
  // making unique_ptr with default_delete the same size as pointer
  class _UniquePtrImpl : private _DeleterT
  {
  public:
    constexpr _UniquePtrImpl() noexcept = default;
    // some other constructors...
    deleter_type& _Deleter() noexcept  { return *this; }
    const deleter_type& _Deleter() const noexcept { return *this; }
    pointer& _Ptr() noexcept { return _MyPtr; }
    const pointer _Ptr() const noexcept { return _MyPtr; }
  private:
     pointer   _MyPtr;
  };
  _UniquePtrImpl   _MyImpl;
};
```

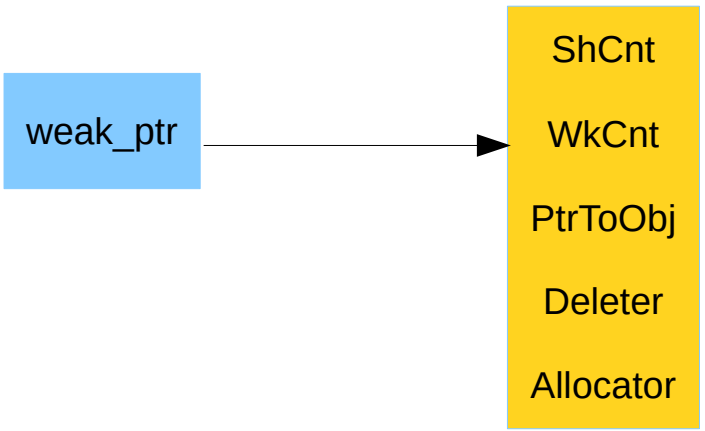
Egy tipikus shared_ptr megvalósítás



Egy tipikus shared_ptr megvalósítás

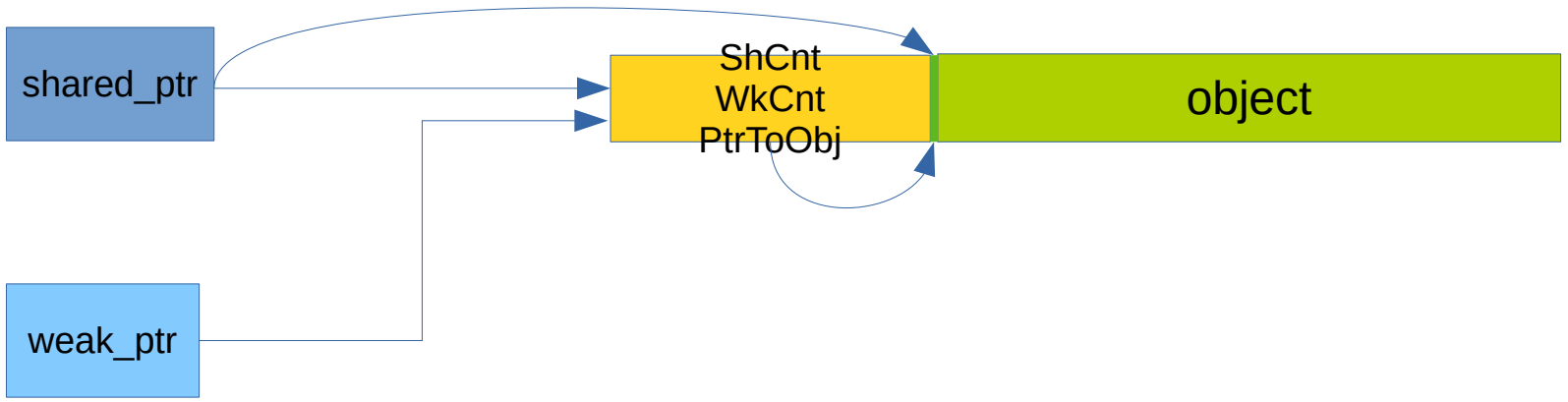


Egy tipikus shared_ptr megvalósítás

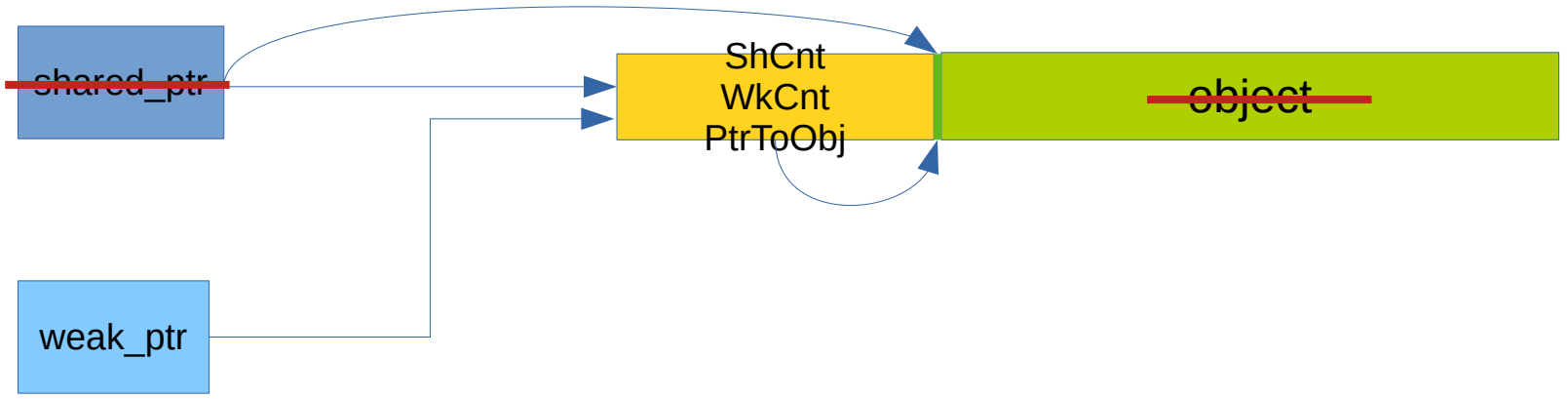


Make_shared megvalósítás

```
// default constructor of T
std::shared_ptr<T> v1 = std::make_shared<T>();
// constructor with params
std::shared_ptr<T> v2 = std::make_shared<T>(x,y,z);
// array of 5 elements
std::shared_ptr<T[]> v3 = std::make_shared<T[]>(5);
```

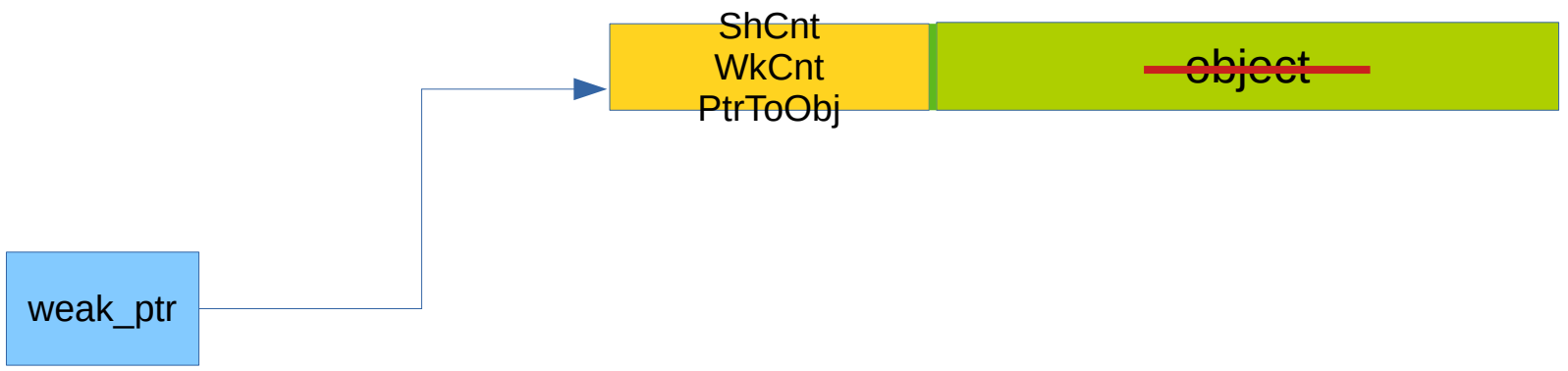


Make_shared megvalósítás



Make_shared megvalósítás

- Az objektum tárterülete nem szabadulhat fel, amíg van élő weak_ptr
- „Kvázi” memory-leak



C++17 párhuzamos STL

- Döntően az Intel Threading Building Blocks (TBB) alapján tervezték
- A legtöbb STL algoritmus kapott egy execution policy paramétert
  - seq, par, par_unseq, unseq (C++20)
- Ezek csak ajánlások, az implementáció szabadon dönthet
- A programozó feladata biztosítani a versenyhelyzet és a holtpont elkerülését
- A minimális követelmény a forward iterátor
- Bevezethetőek újabb policy-k.

Kézi párhuzamosítás

```
// Example from Stroustrup
template<class T, class V>
struct Accum     // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)   // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
    auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
    auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
    auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

Párhuzamos STL

```
// Example from cppreference
template<class T, class V>
struct Accum     // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)
{
    // non-deterministic if binary_op is not associative or not commutative
    double res = std::reduce(std::execution::par, v.begin(), v.end(), 0.0);
    return res;
}
```

Accumulate vs Reduce

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  // fill the vector

  long long sum = 0;
  for ( std::size_t i = 0; i < v1.size(); ++i) // summa x^2  x in [0..49]
  {
    sum += v1[i]*v1[i];
  }

  std::cout << sum << '\n';
  return 0;
}

$ ./a.out
300
```

Accumulate vs Reduce

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  // fill the vector

  long long sum = 0;

  auto sqrsum = [] (auto s, auto val) { return s + val * val; };
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);

  std::cout << sum1 << '\n';
  return 0;
}

$ ./a.out
300
```

Accumulate vs Reduce

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  // fill the vector

  long long sum = 0;

  auto sqrsum = [] (auto s, auto val) { return s + val * val; };
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);
  std::cout << sum1 << ' ' << sum2 << '\n';
  return 0;
}

$ ./a.out
300 300
```

Accumulate vs Reduce

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  // fill the vector

  long long sum = 0;

  auto sqrsum = [] (auto s, auto val) { return s + val * val; };
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);
  std::cout << sum1 << ' ' << sum2 << '\n';
  return 0;
}

$ ./a.out
30000 30000
```

Accumulate vs Reduce

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  // fill the vector

  long long sum = 0;

  auto sqrsum = [] (auto s, auto val) { return s + val * val; };
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);
  std::cout << sum1 << ' ' << sum2 << '\n';
  return 0;
}

$ ./a.out
30000000 59820950156796
```

Accumulate vs Reduce

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  // fill the vector

  long long sum = 0;

  auto sqrsum = [] (auto s, auto val) { return s + val * val; };
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);
  std::cout << sum1 << ' ' << sum2 << '\n';
  return 0;
}

$ ./a.out
30000000 59820950156796
```

Accumulate vs Reduce

```cpp
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  // fill the vector

  long long sum = 0;

  auto sqrsum = [] (auto s, auto val) { return s + val * val; };
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);
  std::cout << sum1 << ' ' << sum2 << '\n';
  return 0;
}
```

reduce() is non-deterministic if binary_op is not associative or not commutative

```
$ ./a.out
30000000 59820950156796
```

Use transform_reduce()

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<long long> v1;
  // fill the vector

  long long sum = 0;

  auto sqrsum = [] (auto s, auto val) { return s + val * val; };
  auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
  auto sum2 = std::transform_reduce(std::execution::par,  // map-reduce
                                    v1.begin(), v1.end(), 0LL, std::plus<>(),
                                    [](auto v) { return v*v; });
  std::cout << sum1 << ' ' << sum2 << '\n';
  return 0;
}

$ ./a.out
30000000 30000000
```

Összefoglalás

- A C++ rohamosan fejlődik
  - 3 évente jönnek ki újabb verziók
  - Könyvtárbővítések
  - Időnként nyelvbővítés
- A modern C++ jobb, mint a klasszikus ...
- … de nem tökéletes :)
- Az egyre több nyelvi és könyvtári elem néha kellemetlenül interferál
- Folyamatosan kell követni a híreket, blogokat, konferencia-videókat

Referenciák

The C++ standard committee site: http://isocpp.org/
Andrew Sutton on Concepts: http://isocpp.org/blog/2013/02/concepts-lite-constraining-templates-with-predicates-andrew-sutton-bjarne-s
Bartłomiej Filipek: How to Stay Sane With Modern C++ https://dzone.com/articles/how-to-stay-sane-with-modern-c
Thomas Becker on Move semantics: http://thbecker.net/articles/rvalue_references/section_01.html
David Abrahams on RVO and Move semantics: http://cpp-next.com/archive/2009/08/want-speed-pass-by-value
Scott Meyers: Effective Modern C++, O'Reilly Media Inc., 2014
Herb Sutter: Back to the Basics! Essentials of Modern C++ Style, CppCon 2014
Mikhail Matrosov: C++ without new and delete, C++Russia, 2015
Nicolai Josuttis: C++17 – the biggest traps https://www.youtube.com/watch?v=h-zy1hBqT74
LLVM/Clang: http://clang.llvm.org/
LLVM/Clang static analyzer: http://clang-analyzer.llvm.org/

Csapdák és buktatók a modern C++-ban

Porkoláb Zoltán
Ericsson Magyarország Kft.
http://gsd.web.elte.hu/
zoltan.porkolab@ericsson.com

Köszönöm a figyelmet!
Kérdések?